International Journal of Advances in Electronics and Computer Science, ISSN(p): 2394-2835 Volume-5, Issue-6, Jun.-2018
http://iraj.in

IMPROVING SEARCH PERFORMANCE OF B-TREE ON FLASH DEVICES

[1]KARTHIK V, [2]G.N. SRINIVASA PRASANNA

[1]MS (by Research) Student, International Institute of Information Technology, Bangalore 560100
[2]Professor, International Institute of Information Technology, Bangalore 560100
E-mail:[1]karthik.v@iiitb.org, [2]gnsprasanna@iiitb.ac.in

Abstract - Flash memory comes with some challenges like write endurance and write amplifications. In this work, we explore a new perspective in implementing B-trees on flash. Previous works have predominantly focused on improving the insert performance, while we consider ways to improves search performance. We propose a novel technique to partition and store newly inserted data and a complimentary concept of metadata to speed up the search performance. The metadata exploits the incremental 1-to-0 update properties of flash in interesting ways to conserve the page invalidations. Our experiments revealed 60% search response time improvement over the best of the alternative schemes.

Keywords - B^+-Tree, Index Structure, Endurance, Flash-Memory, SSD, Storage Systems

I. INTRODUCTION

Due to the rapidly increasing capacity of flash memory and coupled with the declining cost, it is becoming very popular in the design of storage systems and fast replacing the conventional HDDs. The recent addition in this technological advancementsis the 3D NAND flash providing an unparalleled cost advantage of 0.5$/GB and massively dense SSDs with capacities reaching up to 60 TB.

Flash memory response times are significantly lesser than HDDs. However, it demonstrates some distinguishing characteristics/challenges compared to magnetic disks. A flash memory cell supports two type of operations to update its contents: **Program**: Process of applying a voltage to a cell to change its state from the default state (say, 1 to 0). **Erase**: Process of applying a large voltage to a cell to change its state to the default state (say, 0 to 1). The program operation happens at the granularity of a page. The page size is typically 2 KB to 16KB. Erase units are typically large, around 128KB. Erase operations are physically destructive due the application of large voltages. During its lifetime the flash medium can tolerate a finite number of erases (typically, 10^5 cycles).

The traditional block storage devices expose two type of IO operations/interfaces – read/write. But, flash memoryhas a third type of operation–erase (read and program being the other two). This disparity makes it imperative for the file systems or applications directly accessing the flash device to become flash-aware. To avoid this overhead and make flash device also look like block devices to its consumers, an abstraction layer called the Flash Translation Layer [11] is implemented within the flash device. The FTL is mainly composed of three software components: address translation, garbage collector and wear-leveler. The address translation layer translates logical addresses from the file system into physical addresses on flash devices and helps in emulating flash as a normal block device.The layer also performs out-of-place updates which in turn helps to hide the underlying erase operation. The mapping table is stored in a small, fast on-board battery-powered RAM. The garbage collector is in charge of collecting invalid pages to create free space in the flash memory.

Tree index structures are widely used in DBMS [3,15] to improve the speed of data retrieval operations. B-Tree and its variant B+-Tree, have proved to be the most popular tree index structures implementations for HDDs as they can easily accommodate large number of records with a small tree height. Thus, entailing minimal disk accesses for processing lookup and update operations. There is certainly a good opportunity to improve the overall performance of databases and embedded applications alike by persisting the tree index structures from HDD to flash. However, we are posed with certain challenges due to the read/write asymmetry problem on flash. For handling the insert operation into a conventional B-tree node, the key is inserted into its sorted position within the node and updated to the disk. Since, flash does not support in-place update, the current page has to be marked invalid, the modified contents copied to a fresh location on the flash and written/committed. The invalidated page will subsequently be reclaimed by an erase operation. Thus, it can be noted that an update operations on the flash comes with some overheads not present in HDDs which support in-place updates.

A trivial solution to this problem is provided by FTLs [11] which write the modified node to a new unwritten flash page, and hide this low-level page movement by exposing logical page numbers to the index. While the FTL help to abstract out the erase-before-write peculiarities of the flash, it is affected by a phenomenon call write amplification [13], where a

single user write can cause more than one actual write, owing to background activities triggered by FTL. Apart from reducing the endurance of the flash medium, this approach also severely degrades the performance of the device (sometime even lesser than HDDs as explained in [13]), which was our original motivation to choose SSDs over HDDs.

II. PRELIMINARY AND RELATED WORK

Some research work that have been done in implementing B-trees on flash have been presented briefly here:

A. Write optimisations

BFTL [2]: This paper proposes the idea of BFTL (an efficient B-tree layer over the FTL) which is implemented in the file system layer of the operating system. Repeated page writes to the flash memory are processed by modelling new additions/deletions as "index units" and storing them separately. An index unit is treated as a modification of the corresponding B-tree node.A B-tree node is logically constructed by collecting and parsing all relevant index units. Basically, modifications to the nodes are cached in main memory (referred to as reservations buffers). When the buffer fills up, the buffered data is written to the flash via a pre-defined commit policy. The index units of a B-tree node are invariably scattered over flash memory because of the commit policy. Therefore a node translation table is adopted to maintain a collection of the index units of a B-tree node so that the collecting of index units is efficient.

The number of sectors in each logical node is limited by the compaction parameter C. If the number of sectors for each node becomes greater than C, then the BFTL performs the compaction operation that gathers and rewrites the index units together after reading all the sectors where the index units are scattered.

Since multiple update operations can be buffered in the Reservation Buffer and can be later flushed into a sector at once, the update operations of the BFTL causes considerably less write operations on SSDs than B+-Tree. For the search operations, however, the BFTL can read C sectors for a node, thus increasing the search cost by a factor of C in the worst case.

Some of the drawbacks of this approach are: Holding significant amount of cached data in main memory is not fault-resilient during power outages. Using index units to reconstruct the logical view of the B-tree negatively affects the search performance.

LA-Tree [5]; The key idea here is to buffer the updates and insert them within the tree in a lazy manner. There will be a flash-resident buffer attached to a group of nodes at every K levels of the tree. So, there is a hierarchy/tree of buffers.

LA-tree, which has nearly the same structure as the B+-tree except the appended write buffer for each subtree. The basic strategy of deferring node-write operation is similar to that of the BFTL. However, the write buffer is not an in-memory structure like the Reservation Buffer of BFTL but it is stored on the flash. Whereas the Reservation Buffer (of BFTL) globally buffers all the update operations, LA-tree has a write buffer for each subtree that absorbs the update operations coming from the parent node of the subtree. The deferred update operations in the write buffer are dealt in the corresponding subtree, delivering them to the write buffers of the child subtrees, when the write buffer becomes full. This is cascaded till the leaf node.

To balance the search performance and also handle the out-of-place update overhead for flash, the contents of the buffer are pushed down to the lower layers/nodes based on a ADAPT algorithm. This algorithm weighs the cost of emptying the buffer v/s the look-up cost. Based on the outcome, it continues to hold the buffer or flushes the buffer.

Here the buffer is not kept sorted. So, still we need to do a linear search on the buffer which impacts the search performance matching the traditional B-tree. The rampant out-of-place updates of a B-tree arising on flash due to key insertions is contained well but every insert could potentially lead to a write on the flash medium (i.e. the buffers have to be committed into the tree) if the workload influences the ADAPT algorithm to do so. This obviously doesn't help to conserve the life of the costly flash.

The above schemes fall into a category called delta-based schemes. As indicated earlier, the update/overwrite operation on a flash page comes with a performance/endurance cost. So, these schemes try to conserve the writes on tree index nodes by buffering (the delta) them and not overwriting the original node. So, the overall tree comprises the original nodes plus some deltas associated with it. This optimizes the writes, but at the cost of reduced search performance – because all the deltas have to be scanned along with the original node.

B. Search/lookup optimisations

In order to improve the search performance also, the EWOM described in [1], proposes the concept of storing the sorted information about the delta at regular instants. This helps to improve the search performance but still falls behind the B+-Tree because the sorting information may not cover the entire tree and there is a hierarchy of arrays containing sorting information that has to be processed.. We refer to this as a delta-metadata scheme.

In this paper, we propose a metadata-based-B+-tree for flash, which provides a search performance that is close to or matches the B+-Tree and also has an

optimized-write behavior on par with the previous work. The idea makes use of the concept of partial page writes where a portion of a page could be programmed without updating the entire page. The metadata helps to keep the node organized in a sorted manner at all times similar to a B+-Tree node though the new updates/writes to the node are not process like in conventional B+-Tree.

III. FLASH-FRIENDLY B-TREE

The Flash-friendly B-tree structure is similar to a B-tree, but with a new way to organize the leaf nodes (here the large majority of changes happen) in a flash-friendly manner. We assume a leaf node can hold L entries, with the requirement that leaf nodes be at least half full [1] and sorted.As new keys are inserted, if we continue to store them in their sorted positions (like done on traditional B-trees implemented on HDD), it results in a page invalidation on each insertion.Proposed idea tries to solve this problem by building a metadatain a simple and optimal manner. However, while do so, we also do not want to disturb the search performance. Hence, the inserted keys are organized/stored in their respective sorted ranges. Thegeneric model of the Flash-friendly B-treeleaf node is illustrated in Figure 1.

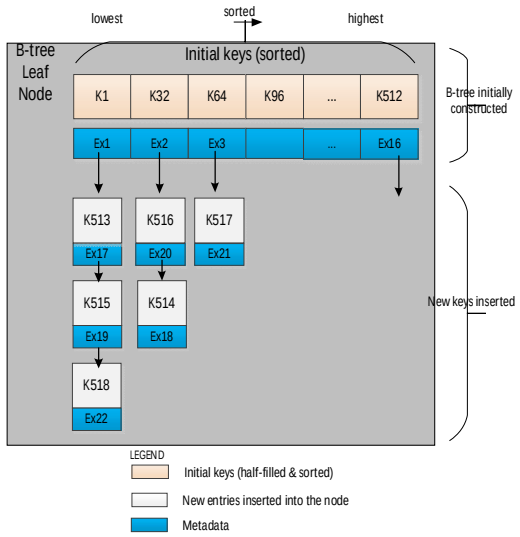


Figure 1Flash-friendly B-tree leaf node.

Data Structures

The proposed flash-friendly version of the B-tree is comprised of some important data structures explained below:

1. **Sorted List (SL)**: The leaf node portion will be initially half-filled (at least) and sorted. The sorted keys implicitly indicate and partition the key value range into Sorted Ranges (**SR**).

2. Sorted Ranges (SR): These are logical portions/buckets. The beginning and ending key values in a contiguously choses set of keys (where each set has identical number of keys) demarcate a specified range of values. In the example (and rest of the explanation) we have chosen L/32 ranges. So, with L = 32, every set of 32 contiguouskeys in SL demarcates the boundary of the SR. If we chose L/16 then every 64 key in SL demarcate the SR and so on. Referring to Figure 1, range 2 has reference to all new keys whose value is >K32 and <= K64. The SR (which is logical) are made up of a linked list of **Key Nodes(KN)**(not to be confused with leaf nodes in a B-Tree).

3. Key Node (KN): The key node is a contiguous memory consisting of a newly inserted key node plus the Extension Pointer. In figure 1, the newly inserted key K513 and Ex1 together form one unit of KN.

4. Extension Pointer (EP): These are flash-friendly version/equivalent of the Next Node Pointer field used in a traditional linked list. The EP is initialized to 0xFFs (which indicates a NULL) and the updated only once with the actual value. Thus the update involves only 1-to-0 transitions. The EP do not hold memory addresses but an ID or offset which uniquely identifies the location of a Key Node within the leaf node. In figure 1, the entities indicated by Ex1, Ex2 so on are the EPs.

Memory Allocation: The capsule of KN consists of data (i.e. key) plus EP i.e. metadata. Leaf node is statically reserved with sufficient space in order to accommodate L/2 keys plus L/2 KNs. In the example used in Figure 1, size of KN would be 4 + 2 = 6 bytes (assuming the size of key is 4 bytes and EP is 2 bytes). So, the EP could address 512 KNs.

Operations on Flash-friendly B-tree

Search: The key is first searched in SL using binary search. If it's unsuccessful, we continue searching within the associated SR (i.e. range with the value range into which the key to be searched belongs). Typically, the range should have few keys (due to balanced partitioning) which can be quickly searched using linear search.Thus, the search is very efficient as we don't need to deal with large unsorted buffers which was the case with prior art [1] [5].

Write Endurance

With the conventional implementation of the B-tree used on a flash memory, there will a need for a page re-write after every key insertion in the tree. The writes to the flash have to be conserved to improve the lifetime and performance of the device which is one of the key objectives of our work.Weachievethis at two levels. First, at the algorithm/data structure level by conceptualizing a metadata that can be incrementally updated and second at the physical/device level by making use of the partial programming characteristics.

There are raw flash implementation which support a concept of partial programming i.e. a page could be programmed in parts. So, a flash of size 2K can be programmed separately in chunks of 512 bytes. This concept can be exploited if the data stored in the page is structured in a manner that it can be updated by appending it in chunks. As mentioned in [11] partial

programming also allows writes to a page, as long as they only involve 1-to-0 transitions.

IV. SIMULATIONS AND EXPERIMENTAL RESULTS

For the implementation a suitable flash simulator [10] was chosen. Discrete objective measures as below were defined and used in order to measure/compare the effectiveness of the algorithms:

- **No of comparisons**: Every key comparison is tracked as a counter which helps to get an idea of the computational complexity of the algorithm,
- **Response time**: This is the end-to-end time duration for the algorithm to complete the operation.

Like any sorting algorithm our algorithm has best/average and worst-cases. The former are typically offline algorithms while we deal with an online problem.

For the purpose of comparative study, we chose the B-tree (as a benchmark), the EWOM algorithm described in [1] and implement it on the flash simulator. In our study, we determine the input pattern that generates the best/worst case for each of these algorithms.

Let's take a real example to search a key. For example, the leaf node has 1023 elements and 512 are initially sorted. The no. of comparisons in the respective algorithms is worked out as below

EWOM [1]:

Step 1: Sorted Array (1*512, binary): 9 comparisons

Step 2: L3 Array (1*288, binary): 9 comparisons

Step 3: L2 Array (2*96, binary): 7 comparisons

Step 4: L2 Array (1*32, binary): 5 comparisons

So, totally, 30 comparisons approximately.

Our Algo:

Step 1: Sorted Array (1*512, binary): 9 comparisons

Step 2: Sorted live node (1*16, binary): 4 comparisons

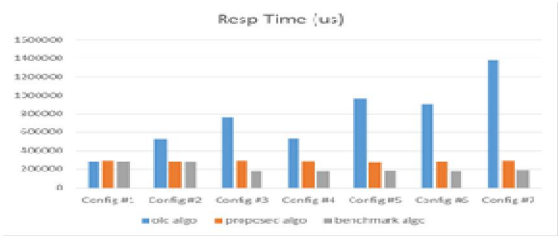
So, totally, 13 comparisons approximately.

A B+-tree was constructed with leaf nodes of different types (i.e. proposed method, EWOM method and a traditional method). The nature of our experimental study was to generate a set of inputs (called configs) that were based on the arrival count (configs 1-7, Table I)

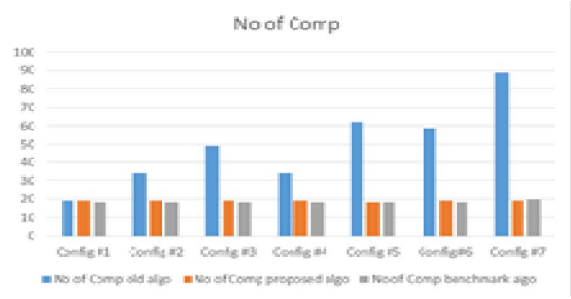
Table 1

Config	Count of L1 Array	Count of L2 Array	Count of L3 Array	Count of unsorted keys (in L1)	New keys/Sorted Key ratio
1	0	0	0	1	1/512
2	0	0	0	16	16/512
3	0	0	0	31	31/512
4	6	2	1	0	288/512
5	6	2	0	0	256/512
6	10	4	1	1	481/512
7	10	4	1	31	511/512

The corresponding results are comparatively in Figure 2 capturing the aforesaid objective measures.



(a)



(b)

Figure 2: Search operations

We can notice (Figure 2) that with progressively EWOM search performance dropped as expected as EWOM as more comparisons were involved while Flash-friendly B-tree performance remained more or less constant/unaffected.

CONCLUSION

We can notice that Flash-friendly B-tree outperformed the prior art in the search category with 60% performance improvement. The Flash-friendly B-tree is search-efficient, simple-to-implement& a generic technique which can be deployed on raw flash.

REFERENCES

[1] Kenneth A. Ross. Modeling the performance of algorithms on flash memory devices. Proceeding DaMoN '08 Proceedings of the 4th international workshop on Data management on new hardware Pages 11-16 ACM New York, NY, USA 2008.

[2] CHIN-HSIEN WU, TEI-WEI KUO and LI PING CHANG. An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems. Journal ACM Transactions on Embedded Computing Systems (TECS), Volume 6 Issue 3, July 2007 Article No. 19 ACM New York, NY, USA.

[3] S.-W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In SIGMOD Conference, pages 55{66. ACM, 2007.

[4] Y. Li, B. He et al. Tree Indexing on Flash Disks. In ICDE 2009.

[5] Agrawal, D. Ganesan, R. Sitaraman, and Y. Diao, "Lazy-Adaptive Tree: An Optimized Index structure for Flash Devices," in Proceedings of the VLDB Endowment, 2009, pp. 361–372.
[6] Hua-Wei Fang, Mi-Yen Yeh, Pei-Lun Suei, and Tei-Wei Kuo. An Adaptive Endurance-Aware B+-Tree for Flash Memory Storage Systems IEEE Trans. Computers 63(11):2661-2673 (2014).
[7] Xiaona Gong, Shuyu Chen, Mingwei Lin, Haozhang Liu. A Write-Optimized B-Tree Layer for NAND Flash Memory. Wireless Communications, Networking and Mobile Computing (WiCOM), 2011 7th International Conference, pages 1-4, 2011
[8] Luis Cavazos Quero, Young-Sik Lee, and Jin-Soo Kim. Self-sorting SSD: Producing sorted data inside active SSDs. MSST, pages 1-7, IEEE, 2015.
[9] M. Athanassoulis and A. Ailamaki. BF-Tree: Approximate Tree Indexing, Proceedings of the VLDB Endowment, Vol. 7(14), 2014.
[10] Flash Simulator, http://csl.cse.psu.edu/publications/flashsim_tr.pdf
[11] Intel Corp. Understanding the Flash Translation Layer (FTL) Speci_cation, 1998.
[12] Deepak Ajwani, Itay Malinger, Ulrich Meyer and Sivan Toledo. Characterizing the performance of Flash memory storage devices and its impact on algorithm design. Proceeding WEA'08 Proceedings of the 7th international conference on Experimental algorithms Pages 208-219, 2008.
[13] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives", in SYSTOR 2009:The Israeli Experimental Systems Conference, 2009
[14] Paul England and Marcus Peinado. System and method for implementing a counter, 2006. US Patent Number 7,065,607.
[15] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin and Sang-Won Lee. B+-tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives. Journal Proceedings of the VLDB Endowment VLDB Endowment Homepage archive Volume 5 Issue 4, December 2011 Pages 286-297.
[16] Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high performance flash disks. In SIGOPS Operating Systems Review, volume 41(2), pages 88{93, 2007.
[17] Myers and S. Madden. On the use of NAND flash disks in high performance relational databases. 2007.

★★★